Projet Compilation

LICENCE INFORMATIQUE

2020-2021

RAVENDRAN | SDAF

SOMMAIRE :

I. PRÉSENTATION

L'objectif de ce projet est de réaliser un compilateur à l'aide de flex et bison pour le langage TPC. La documentation sur la partie Flex et bison fait au semestre 5 se trouve dans **SDAF_RAVENDRAN_Rapport_AS.pdf.**

Nous avons laissé un fichier **script.sh** qui permet de lancer tout les tests des différents sous-répertoire : **correct, erreur sémantique, erreur syntaxique et warnings**.

Pour lancer le programme sur un fichier.tpc, placez vous :

- dans le répertoire bin, et lancer la commande

  ./tpcc [OPTIONS] < ../[LIEN FICHIER .TPC]

  OU

- dans le répertoire du projet et lancer la commande

  bin/tpcc [OPTIONS] < [LIEN FICHIER.TPC]

où

- **LIEN FICHIER.TPC** correspond au chemin du fichier test sélectionné, il existe 4 sous répertoires dans **tests**:
  - tests/**good**/fichier.tpc : good contient des fichiers tpc corrects,
  - tests/**warn**/fichier.tpc : contient des fichiers contenant des warning,
  - tests/**syn-err**/fichier.tpc : contient des fichiers avec des erreurs lexicales/syntaxiques,
  - tests/**sem-err**/fichier.tpc : contient des fichiers qui ont des erreurs sémantiques.
- **OPTIONS** :
  - **-t** : affiche l'arbre abstrait sur la sortie standard,
  - **-s** : affiche toutes les tables des symboles sur la sortie standard ,
  - **-h** : affiche une description de l'interface utilisateur et termine l'exécution.

II. DÉCOMPOSITION DU PROJET

1. ARBRE ABSTRAIT

Lors de l'étude lexicale du fichier étudié, nous avons créé un arbre contenant toutes les instructions du fichiers TPC analysé permettant de stocker les variables, identifiant et autres.

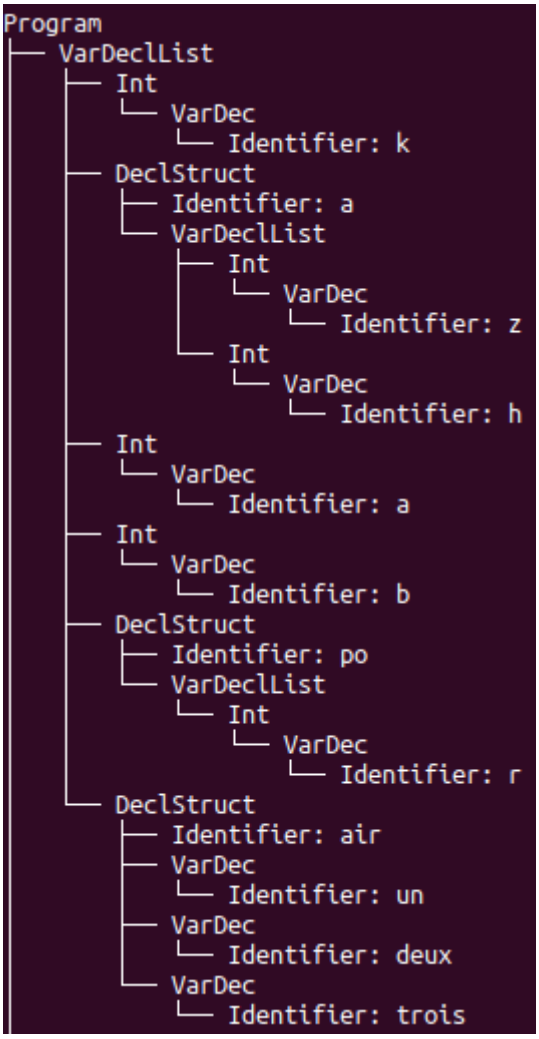
```
int k;
int a;
int b;

int fct_1(void){
    a = 3;

    if(a >= 1 || a >= 1 + 4)
        a = 6;
    else
        a = 15;
    while(a < b && a > 34)
        // Ceci est un commentaire d'une ligne
        b = 9;
    return a;

    return 0;
    {return 0;}

    print(6);


}

int main(void){
    return 0;
}
```

FICHIER .TPC ANALYSÉ.

ARBRE CORRESPONDANT AU FICHIER .TPC.

Si on ajoute des structures dans notre fichier tpc on aura un arbre presque identique, les structures seront afficher au début de l'arbre :

```
int k;

struct a{
    int z;
    int h;
};

int a;
int b;

struct po{
    int r;
};

struct air un, deux, trois;

int fct_1(void){
    a = 3;

    if(a >= 1 || a >= 1 + 4)
        a = 6;
    else
        a = 15;
    while(a < b && a > 34)
        // Ceci est un commentaire d'une ligne
        b = 9;
    return a;

    return 0;
    {return 0;}

    print(6);


}

int main(void){
    return 0;
}
```



ARBRE AVEC LES STRUCTURES DU FICHIER TPC.

2. TABLE DES SYMBOLES

Nous avons donc construit la tables des symboles lors d'un premier parcours de notre arbre. Cela permet de donner des adresses à chaque variables déclarés et aussi de retenir leur type, leur portée, leur nom.. Mais également de vérifier que toutes les variables utilisées sont déclaré. La redéfinition de variables est aussi détecté grâce à cette table.

symbole table

Niveau	Classe	name	pos_line	type	pointeur	nbr arg	adresse
0							
	Global	k	1	Int	0	0	8
	Global	a	4	Int	0	0	16
	Global	b	5	Int	0	0	24
1							
	Fonction	fct_1	7	Int	0	0	0
2							
	Fonction	main	27	Int	0	0	0

TABLES DES SYMBOLES DU FICHIER TPC SANS STRUCTURE.

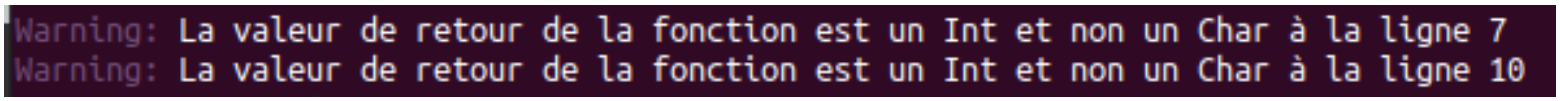
symbole table

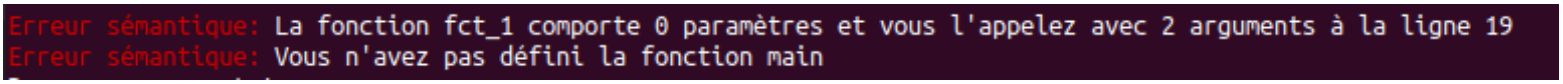
Niveau	Classe	name	pos_line	type	pointeur	nbr arg	adresse
0							
	Global	k	1	Int	0	0	8
	Global	a	8	Int	0	0	16
	Global	b	9	Int	0	0	24
	Global	un	15	DeclStruct	0	0	32
	Global	deux	15	DeclStruct	0	0	40
	Global	trois	15	DeclStruct	0	0	48
1							
	Fonction	fct_1	17	Int	0	0	0
2							
	Fonction	main	37	Int	0	0	0

TABLES DES SYMBOLES DU FICHIER TPC AVEC STRUCTURE.

3. TYPAGE

Une fois la table terminée, les seuls erreurs non vérifiée sont les erreurs de type. Dans cette partie nous avons vérifié que les types lors des assignations, des retours ou appel de fonctions, ect sont corrects. Dans le cas inverse, nous avons levé des erreurs ou des warnings. Les adresses et les pointeurs sont gérés au niveau du typage. Nous avons également profité de cette partie pour vérifier qu'une fonction main existe dans le programme.





4. TRADUCTION

Nous avons terminé ce projet par la traduction. Nous avons réussi à gérer les appels de fonction, les déclarations de variables globales et locales, les readc et reade, les print. Cependant nous n'avons pas eu le temps de traiter les adresses, pointeurs et les structures.
Cette partie est lancé uniquement si le programme ne comporte pas d'erreur.

5. TEST

Nous avons également préparé **quatre dossiers de tests différents** : good, syn-err, sem-err et warn. Dans le dossier good, il y a que des **fichiers tpc correct, le code nasm sera donc créé** (sans les structures). Dans le fichier syn-err, nous avons mis dedans des **fichiers tpc contenant des erreurs, la traduction n'aura donc pas lieu** mais dans **le répertoire sem-err** (avec des erreurs sémantiques) et **dossier warn, les fichiers seront traduit en nasm**.

III. DIFFICULTÉS RENCONTRÉES

Dans ce projet, nous n'avons pas pu le traiter entièrement :

- Dans la partie nasm nous n'avons pas réussi à nommé notre fichier tpc traduit avec son propre nom (ex : file.tpc traduit doit donner file.asm), il aura comme nom de fichier **_anonymous.asm** car la commande bash '<' ne nous permet pas de récupérer le nom du fichier comme argument de notre ligne de commande (On a laissé notre idée de code en commentaire dans le fichier **writeNasm.c** a la fonction **writeNasmFile**),

- Nous avons réussi a retranscrire un arbre avec des structures mais pour la table de symboles c'est plus compliqué : on a essayé de faire en sorte que les structures soient acceptées dans celle-ci mais nous n'avons pas pu résoudre complètement le problème (plusieurs sont en commentaire dans le fichier symboleTable.c et symboleTable.h).